ACM

Symposium on Operating System Principles

October 1-4, 1967 Gatlinburg, Tennessee

AN IMPLEMENTATION OF A MULTIPROCESSING COMPUTER SYSTEM

William B. Ackerman
William W. Plummer
Department of Electrical Engineering
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

Association for Computing Machinery 211 East 43 Street New York, N.Y. 10017

AN IMPLEMENTATION OF A MULTIPROCESSING COMPUTER SYSTEM

William B. Ackerman and William W. Plummer

Department of Electrical Engineering
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

This work was supported by the Joint Services Electronics Program (Contract FR28-043-J6-00495)

1. Overall Structure and Objectives of the System

A PDP-1 computer was donated (by the Digital Equipment Corporation) to the Electrical Engineering Department of the Massachusetts Institute of Technology in late 1961. In May, 1963 the first time-sharing system was operational.

Since 1963 this PDP-1 has undergone substantial modifications (c.f. Appendix). Presently the machine has twelve thousand words (18-bit) of five microsecond memory arranged in pages of four thousand words. One of these pages is reserved for the system code and is protected from user references.

This paper describes a recent multiprocessing system which has been implemented on the PDP-1. The principle design criteria of the system are:

1) that it be modular in the sense that the supervisor may be constructed of independent and asynchronous processes;

2) that I/O functions may be controlled directly by user mode processes; and,

3) that it contain an effective scheme for allocating system resources to computations and providing protection for these resources.

The system is composed of several parts which are:

1) the executive routine which accomplishes process scheduling and implements certain meta-instructions;

2) scheduling hardware that is used by the executive routine. (This is a realization of Corbato's multi-level queue scheduling algorithm. See Reference 3).

3) the supervisor which implements I/O buffering, general purpose routines to be called by users, the file system, and computation control, i.e., scheduling of computations and allocation of resources; and,

4) the user computations, which may include an assembler, a text editor, and a combined monitor and debugging routine.

Schematically, the system appears as shown in Figure 1. Transitions from user processes to the executive routine are caused by traps and interrupts. To resume a user process, the executive routine executes an unbreak instruction, which will be discussed more completely in a later section.

At this point we draw a distinction between traps and interrupts. A trap is an event that occurs in synchronism with the machine. Generally traps are caused by illegal instructions and special instructions which request service by the executive routine. The system meta-instructions, including the process control primitives, are implemented as traps. Interrupts are events which are asynchronous with respect to the computer. Interrupts notify the executive routine of I/O device completion.

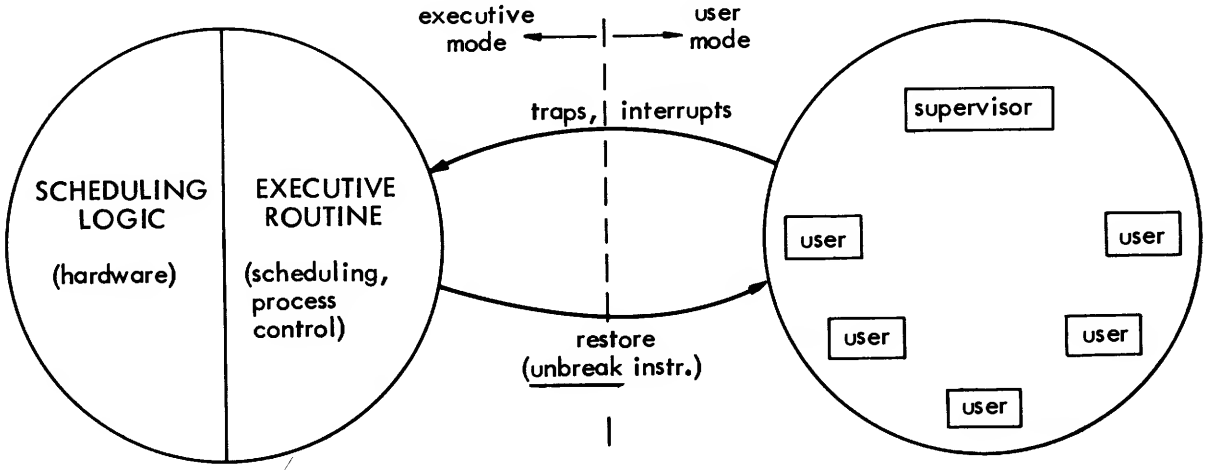

Figure 1 Overall Schematic of the System

2. Multiprocessing Primitives

A process is a virtual processor, the state of which is determined by its live registers. A process will be created when an existing process executes a fork meta-instruction. Fork has the effect of dividing the "flow of control" into two paths - the original process, and the process created by the fork meta-instruction. Because any process may execute a fork, a large number of parallel processes may be established. A process may delete itself by executing a quit meta-instruction.

Quite often it is required that a particular process not be started until some number of other processes quit. This control is achieved by the join meta-instruction. Join takes an argument which is the number of processes to be joined. For all but the last process to execute it, the join behaves like a quit meta-instruction. For the last process, it behaves like a "no operation" instruction.

Because several processes may share common data, it was necessary to provide the lock and unlock meta-instructions. These are used to prevent one process from using data which has been only partially updated by some other process.

Situations arise in which a large number of processes may wish to execute a certain block of code but only a fixed number are allowed to. To handle this situation, the enter queue and release queue meta-instructions are used. When a process attempts to enter the block but is not able to, it executes an enter queue to suspend itself. The release queue meta-instruction indicates to the executive routine that the waiting process may proceed. Enter queue and release queue may be used as generalized lock and unlock instructions.

3. Motivation for the System

One of the principal functions of a time-sharing supervisor is the processing of requests for various types of services from programs running under the system. Timing and protection considerations demand that a large number of operations, such as input-output, access to mass memory, and allocation of resources, be serviced by a supervisory program which is protected from the programs requesting the service. The supervisor handles buffering of I/O devices, to keep them busy even when the program requesting the I/O operation is no longer in core, and protects each user's programs from tampering by other users.

Since several user programs can request the same supervisor function at the same time, (or nearly the same time, if there is only one CPU), there must be some mechanism to prevent conflicts. Where there is only one CPU, and the service takes a very short time, the service routine could run in executive mode, or in user mode in which the scheduler has guaranteed that its quantum will not end. In this way a second request cannot possibly occur until the first one has been serviced. If the supervisory function will take a long time, (for example, if it involves I/O operations), it is necessary to let other programs run before completing the function, during which time other programs can request the same function. A simple example of this is I/O device interpretation and/or buffering. If the device is not ready, the service routine must set some sort of flag indicating that this particular process has been suspended, and then go to the scheduler to run another process. When an interrupt indicates that the supervisory function is able to continue, the interrupt handling routine transfers back to the service routine.

The transfer to the process scheduler and return from the interrupt handler are simply a request to suspend the supervisor process until the I/O device completes, but it requires direct connections between the program service routine and the scheduling part of the executive. These connections become more complex as the manner in which the service routine uses the I/O device becomes more complex.

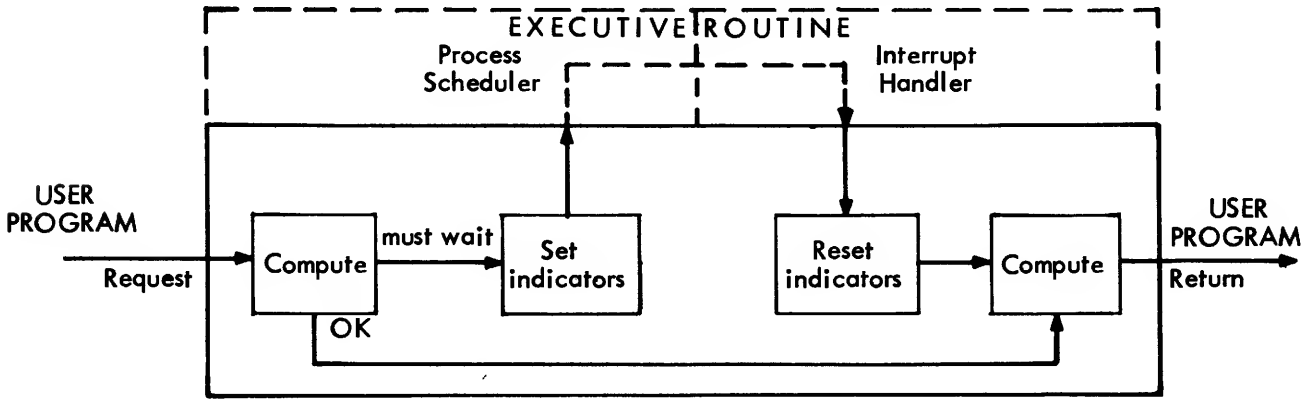

Figure 2 Primitive IO Request Handler

This I/O supervisory routine is reentrant in a primitive way. Several different requests for the same function can be in progress at one time, but only if they are waiting in the process scheduler part of the routine. This method of processing requests for action by the supervisor requires that all supervisory routines either complete in a very short time or wait for some external event to restart the service routine. The routine to process such a wait must be connected to the process scheduler, and the data pertinent to the wait must be private to each waiting process.

These problems may be avoided by making the routine run, in user mode, under the scheduler. The entire routine must be reentrant, and uses a supervisor call to the scheduler to request a wait for I/O device completion. This requires that the time-sharing system have three levels. The highest level is the executive routine, which runs in executive (non-interruptible) mode and handles process scheduling. The second level is the supervisor, which runs in user mode and processes requests from user programs for various services. The third level consists of the user programs themselves. Both the supervisor and the user programs are scheduled by the executive routine.

When a request from a user program to the supervisor involves operation of an I/O device, the I/O instruction is done in the supervisor, in user mode. When the device completes, the supervisor must be informed of that fact in order to operate the device again and/or return to the user. For this purpose, initiation of any I/O operation causes a function started trap to the executive routine. The scheduler suspends the process that caused the trap and runs some other process. When the device completes, a function completed interrupt occurs, and the scheduler places the waiting process back on the queue of processes to be run. In this way the request for a wait until an I/O device completes is always associated with the initiation of an I/O operation, and vice-versa. (Some I/O operations complete immediately. In this case there is no trap or interrupt, and the program continues in user mode.)

Where the supervisory function involves I/O buffering, it is frequently necessary to return to the calling program immediately even though the actual device is still busy. This requires the use of the process control meta-instructions fork and quit. An example of an I/O buffering program will be treated in detail later.

Processes running in user mode (in the supervisor or in actual user programs) can also request a wait for an event that occurs in another process, rather than in an I/O device. The meta-instruction enter queue requests such a wait, and release queue, executed later by another process, restarts the waiting process.

All of the executive service functions, such as fork, quit, and enter queue, are available to user programs as well as the supervisor, and the facilities for writing user mode service routines are also available, so that user computations can define service functions for inferior computations.

The structure of the system, with respect to the service functions provided, is shown below.

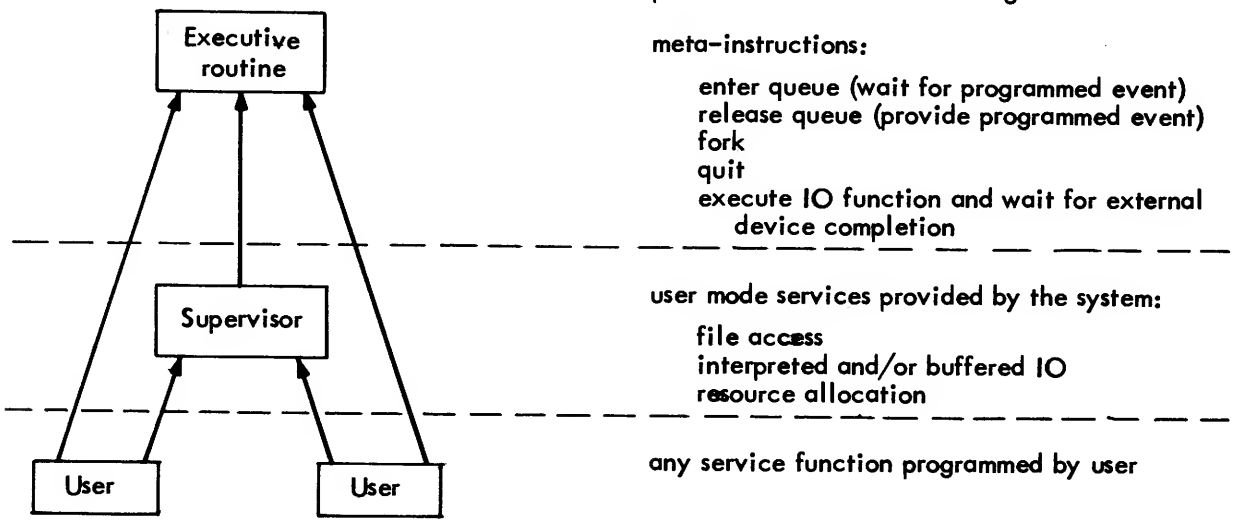

Figure 3 System Services

4. Hierarchy of Computations

A computation is a block of virtual memory in which processes may run. A computation is also the basic unit of protection, since every process running in a computation is bound to that computation. Processes may execute instructions affecting I/O devices, mass memory and other computations only if the capability list, or C-list, of that computation permits.

The supervisory computation and user computation(s) may create inferior computations. Thus, each computation (except the supervisor) has an immediate superior, another computation, to which protection violations by processes in the first computation are reported. The immediate superior owns the computation, and has an appropriate entry in its capability list. All of the computations in the system thus form a tree structure of ownership and superiority, with the supervisor at the top.

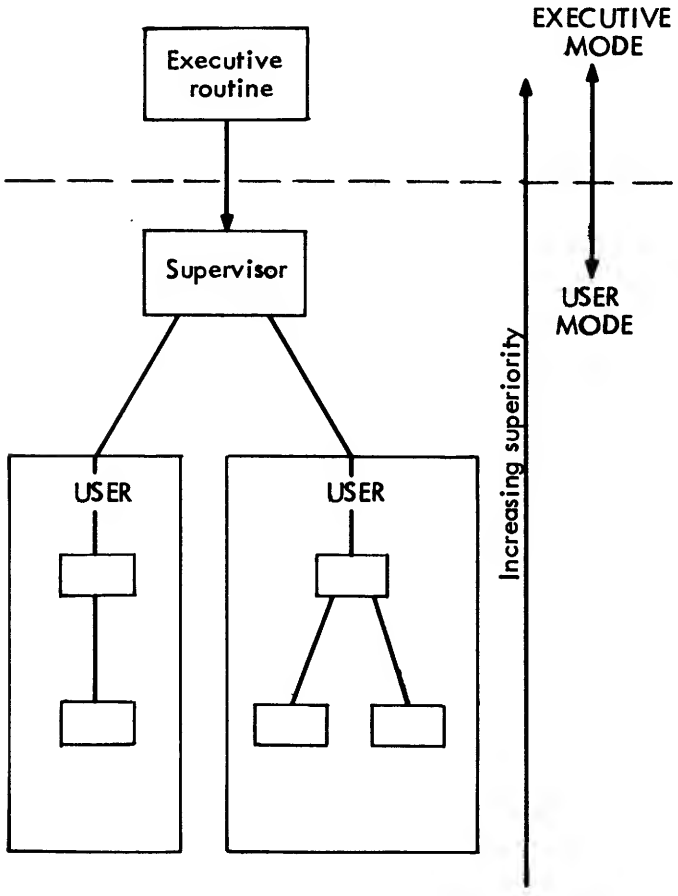

Figure 4 The Hierarchy of Computations

5. Protection and Capability Lists

Protection is implemented (in addition to the usual hardware detection of privileged instruction violations, etc.) by a capability list ("C-list") associated with each computation. The C-list of a computation indicates which of certain system functions and resources are available to processes running in that computation. The types of capabilities are:

inferior computation
suspended process
I/O device
file
directory
programmed queue
entry

The invoke meta-instruction, executed by any process in a computation causes a capability to be invoked. The action taken depends on the capability and parameters supplied by the process.

The computation hierarchy is built out of inferior computation capabilities. An inferior computation capability in the C-list of computation X indicates that the computation specified in the capability, say Y, is inferior to X. X exercises complete control over all aspects of Y, and is protected by both hardware and software in the executive routine and supervisor from the effects of any malfunction by processes in Y. Processes running in computation X may grant and revoke (with meta-instructions bearing the same names) capabilities in Y's C-list. Any processing fault (illegal instruction, attempt to tamper with the protection mechanism, etc.) by processes in computation Y start a process in computation X, beginning at the fault entry address, reporting the malfunction.

Computations may define service subroutines to be called by other computations by creating an enter capability and granting it to the other computations. The capability specifies the computation to be entered and the starting address. When a process invokes an enter capability, that process is suspended (by the executive routine) and a new process is started in the computation being entered. A suspended process capability is created in the C-list of the entered computation and the service routine (in the entered computation), may, by invoking this capability, examine and modify the registers of the calling process and do whatever is necessary to process the request, finally executing a proceed meta-instruction to resume processing by the calling process. The calling process may transmit a capability from the C-list of its own computation to the service routine. This is useful for certain types of service routines, particularly those involving I/O operations. A service routine of this type may be reentrant and hence may be simultaneously servicing requests from several

processes, although the process locking techniques to be discussed later may be used to limit the number of processes permitted in certain sections of the routine at any one time. The ability to have reentrant service routines is especially useful for coding routines that take a long time, such as a routine to take input from a typewriter and put it in a file.

Most of the usual supervisor functions, such as I/O buffering and file management, will be called by enter meta-instructions.

A file capability is a special case of an enter. It was designed to be a separate capability only to economize on the amount of information that needs to be specified in the capability. The file processor is a service routine in the supervisor.

A directory capability is also a special case of an enter. A directory is a file containing associations of symbolic names and capabilities of any type. A directory is a convenient way of storing a large number of capabilities, of transmitting many capabilities from one computation to another, and of storing a list of named files. (A file capability alone does not have a name associated with it.)

A programmed queue capability actually has very little to do with protection and is put in C-lists only for convenience. It is discussed in the section on process control.

The three basic functions of any time-sharing supervisor are to schedule the execution of several programs, to protect programs from adverse effects of malfunctions in other programs, and to provide service routines for the programs. The PDP-1 time-sharing system makes all three of these facilities available to user programs. Hence, it is possible for a user program to simulate a complete time-sharing system, using enter to simulate all invokes and meta-instructions. Although this is not contemplated, enter is an extremely useful tool for extending the power of the other capabilities. For example, to make an assembler take its source text from something other than its input file, an enter capability may be substituted for the file capability expected by the assembler. Every attempt by the assembler to read from its input file will instead transfer control to a service routine.

6. Input-Output

A process may invoke (initiate) an I/O function. If this happens, the process will be suspended until the I/O function completes. At a given time there may be several processes operating different I/O functions, as well as ones which are simply computing. Two processes attempting to operate the same I/O function is handled as an error condition.

It is useful to compare the effect of a completion interrupt in this system with effect in the more conventional "sequence break" or "program interrupt" organizations. When an interrupt occurs in a sequence break system, it causes the computational level program to stop, and an I/O service routine to start. This routine is responsible for determining which device completed, restarting it, and returning to the main program. In the PDP-1 multiprocessing system an interrupt from a given function simply restarts the process which invoked that function.

The real time clock is an I/O device available to user mode programs. This permits the computation scheduler to be run as a user mode process which does the drum operations necessary switch computations and resets the clock. The setting of alarm times for all user processes is handled by another supervisory routine which keeps a queue of alarm times and updates the actual alarm time when required.

7. Reentrant Programming and Process Control

In order to properly write reentrant programs, it is usually necessary to have a larger amount of private data than is provided by the live registers alone. An index register was constructed principally to make reentrant programming practical. By having each process's index register point to its own block of working storage, several processes may execute the same instructions and use the same constants while referencing different areas of working storage.

A fundamental operation is the locking out of other processes, preventing more than one (or some other number) of processes from executing a certain block of instructions at one time. At the beginning and end of the block of code there are instruction sequences to effect the lockout. A simple example of this is a program to modify a file or some other data object, in which the result is not usable until the operation is complete.

In the general case, the block of code is preceded by a lock meta-instruction and followed by an unlock meta-instruction, the instructions each having the address of the lock indicator for the specified block of instructions. Lock could be a hardware instruction or a meta instruction serviced by the executive routine. A process executing the lock instruction is permitted to proceed only if the lock indicator is off, and, if so, the lock indicator is turned on. The test and set operations must be performed together without the possibility of intervention by another process or processor. The unlock instruction would clear the lock indicator.

In the present system, only one processor exists so only one process will execute the lock at one time. Assuming the block of locked instructions will not produce a trap (due to illegal instruction, etc.), it was sufficient to realize lock as a machine instruction which inhibits interrupts, which would cause a different process to be started. Unlock enables

interrupts after a lock. A lock timer produces an error condition if a process tries to remain locked for more than sixty-four memory cycles.

Where the block of code in which only one process is permitted is more than sixty-four memory cycles in length, or where the desired synchronization is more complex than just restricting something to one process, the instructions enter queue and release queue are provided. The enter queue meta-instruction suspends the process executing it. Any other processes are permitted to continue, so a process may remain suspended for any length of time without disrupting the rest of the system.

To restrict the number of processes entering a (possibly very long) block of code, a count and test routine is placed at the beginning of the program. Only the first process is permitted to go through. (This count and test routine must be executed in one instruction or else be locked. Enter queue is effectively a single instruction.) All later processes will execute the enter queue and be suspended. During the entire period that the block of instructions is being executed by the one process that was permitted to enter it, other processes elsewhere in the system are permitted to run. Only processes that attempt to enter the same routine are suspended. When the process exits from the block of instructions, it clears the flag indicating that a process is in the block and executes a release queue addressing the same queue. If no processes were suspended in the queue, the release queue does nothing. If any processes were suspended, the first one that entered the queue is removed from the queue and restarted in locked mode. Between the time that the flag is cleared and the release queue is executed, the computer should be locked. This, along with the fact that the restarted process is locked insures that the restarted process will find the flag off and will be permitted to enter the restricted block of code. Since the first process to execute the enter queue is the first to be released, processes are queued in an equitable way.

Enter queue and release queue may be used for programming more sophisticated process control operations. For example, where a limited number of storage areas are available for a reentrant routine, it is necessary to restrict the number of processes entering the routine to that number, and to point each process's index register at its storage area.

In some cases, a simpler type of process control may be accomplished by using quit to stop a process and fork to recreate it. That would not work in the above case if the processes entering the routine have private data in their live registers because these data are lost at a quit (and, until a process has been through the queue, it cannot allocate any memory in which to store its registers). Live registers are preserved in a queue. This simpler method of starting and stopping processes may be applied to I/O buffering routines.

The example shown in Figure 7 is typical of the supervisory procedure for some I/O devices. This is a routine to buffer typewriter output, which accepts calls from user computations and places the information in a buffer, returning to the calling program immediately unless the buffer becomes full, in which case it fails to return until the buffer is nearly empty. To properly understand this example, it must be remembered that a process which executes an I/O function is suspended until the function completes.

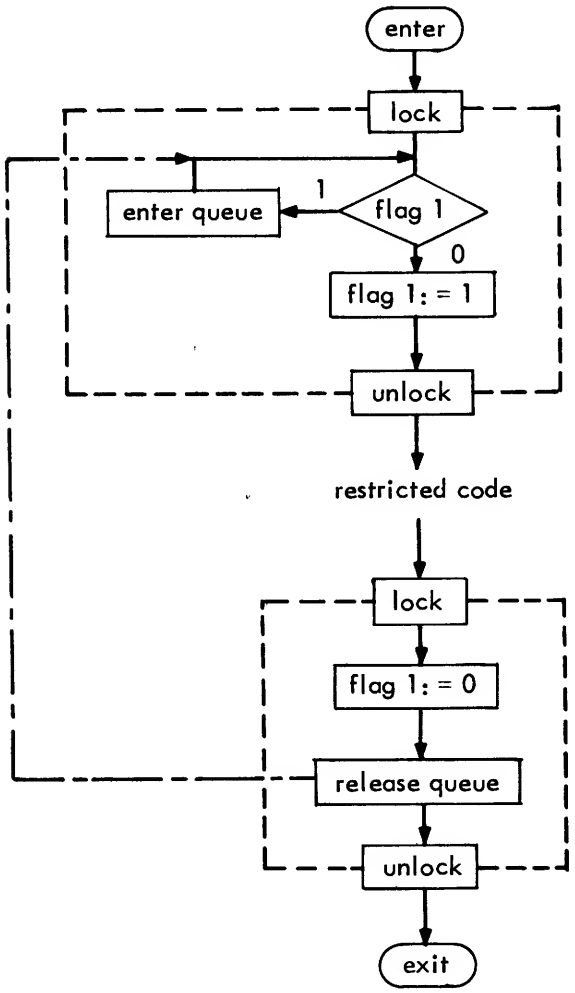

Figure 5 Programmed Lockout

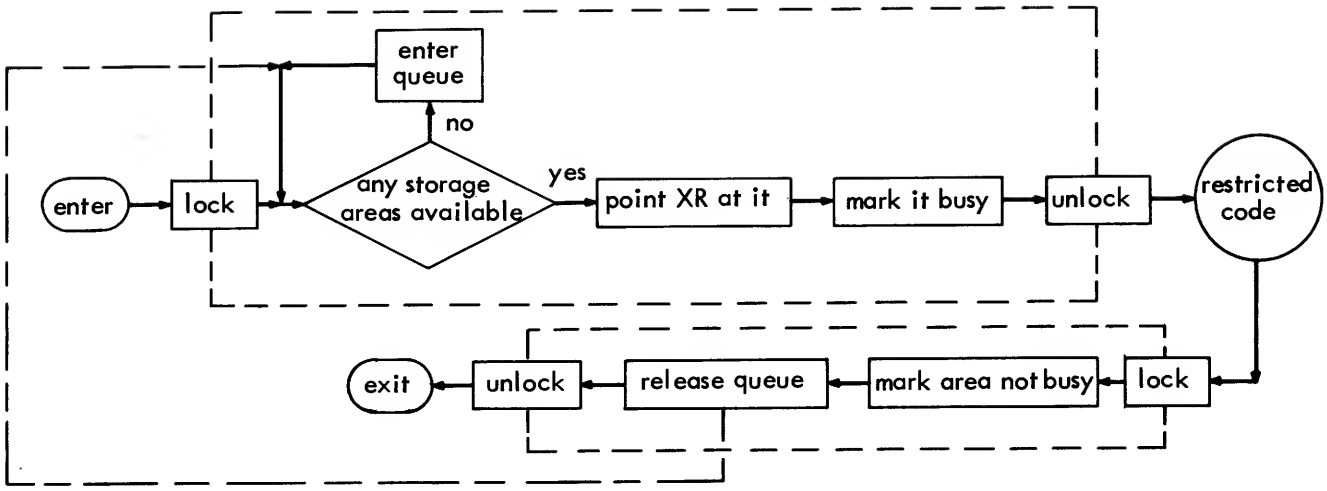

Figure 6 Allocation of Storage Areas

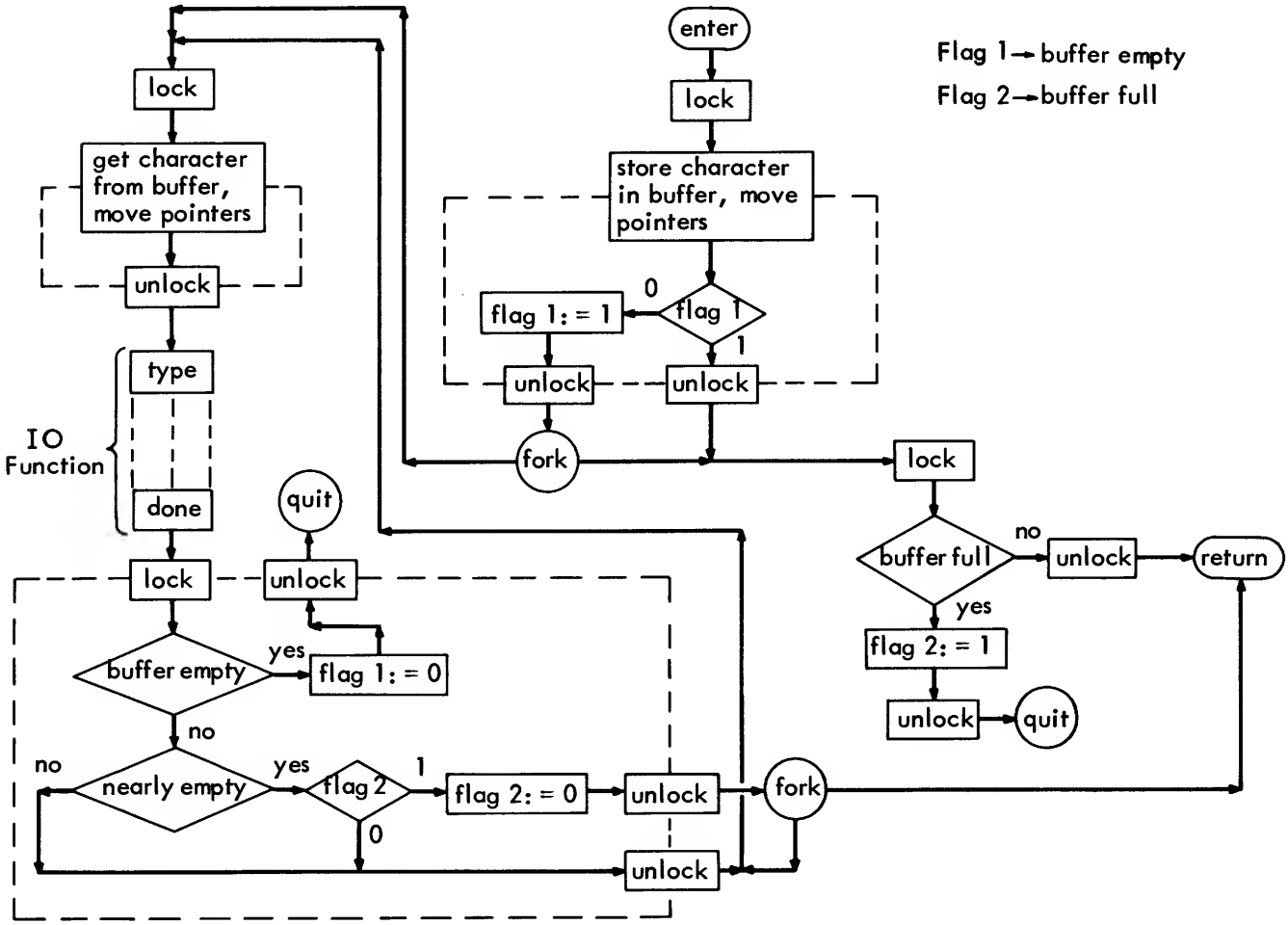

Figure 7 Sample IO Buffering Routine

8. Modes, Traps, and Interrupts

When the time-sharing system is running the computer is in one of two modes, user mode or executive mode. The executive routine runs in executive mode, and all other routines, including the supervisor, run in user mode. When the computer is in user mode, any I/O device completion, process quantum end, illegal instruction, or execution of a meta-instruction or invoke causes an interrupt or trap. The computer switches to executive mode, automatically stores all live registers, and resumes operation at a location in the executive routine peculiar to the reason for the interrupt or trap. The executive routine can return to user mode and restore all live registers to their state prior to the trap or interrupt by execution of the unbreak instruction. This instruction is privileged, that is, legal only in executive mode. To facilitate switching from one process to another, the executive routine can specify the locations in core memory in which the live registers are stored by loading an address in the process pointer register. The unbreak instruction loads the computer's live registers from consecutive locations beginning with the location addressed by the process pointer. While the computer runs in user mode, the process pointer remains unchanged. (The instruction to load the process pointer is privileged.) When a trap or interrupt occurs, the computer's registers are stored in the locations from which they were loaded, as directed by the process pointer. To switch processes, the executive routine simply loads the process pointer with the address of the stored registers of the new process and executes an unbreak instruction.

9. The Process Queue

There is an eight level process priority queue containing all processes which are in core and runable at a given time. The process scheduling algorithm is to run the first process at the highest occupied queue level. When its quantum terminates, (assuming that it has not already terminated processing for some other reason) it is placed at the end of the queue at the next lower level, or the same level if it is already at the lowest level. The first process on the highest occupied level is then run. When a level becomes empty, the process scheduler runs processes at the next lower level. In the absence of I/O operations, all processes will eventually descend to the lowest level. A process which is restarted because of I/O completion is placed on a level determined by the device.

The hardware scheduling logic includes registers indicating the priority level of the running process and the highest priority level of processes which are not running. These registers are loaded by the executive routine and determine the operation of the process quantum clock.

A process quantum varies from 1 1/4 ms. for the highest level to 40 ms. for the lowest level, and is always a multiple of 1 1/4 ms., the subquantum time. At the end of each subquantum, the hardware checks the number of subquanta remaining in the quantum, the level of the current process, and the level of the highest process remaining in the queue. If a process of higher priority has appeared in the queue and the quantum of the current process has not ended, a preempt interrupt occurs, and the process scheduler replaces the interrupted process at the head of its queue level, to run for the remainder of its quantum at a later time, and the higher priority process is run. If the quantum of the current process ends and there is another process of equal or higher priority, a round robin interrupt occurs, and the process scheculer places the stopped process at the end of the queue at the next lower level. If the quantum ends and there is no process of equal or higher priority, the same process would run next anyway, so the hardware automatically demotes it to a lower level, and no interrupt occurs.

When a process executes an I/O instruction and the device does not complete immediately, the I/O function started trap occurs. The executive routine suspends the process by removing it from the queue, and runs the process of highest priority. When the device completes, an I/O function completed interrupt occurs, and the process is replaced on the queue. It is not run immediately, but if its priority is sufficiently high, it will cause a preempt interrupt and will run next (within 1 1/4 ms.).

An enter or similar instruction requiring service by a user mode computation causes a trap to the executive routine. The executive routine creates a process in the computation being entered and schedules it to be run. The original process is suspended by being removed from the queue.

An illegal instruction is treated like an enter, and a new process is started in the immediate superior to the computation in which the instruction was executed, starting at the fault entry address. The supervisor has no superior, and an illegal instruction trap from the supervisor is considered a system failure.

Certain supervisory functions are processed directly by the executive routine. These include fork, quit, enter queue, and release queue, which require the non-interruptibility of the executive routine in order to function properly. When a fork trap occurs, the executive routine allocates a block of memory for the stored registers of the new process and executes a special instruction which switches to user mode and automatically copies the stored live registers of the old process into the new one. The routine to service the quit trap returns the block of memory used by the process to the list of free process blocks and runs some other process. Enter queue places the process at the end of the

programmed queue and removes it from the main process queue. Release queue removes the first process in the programmed queue and places it at the head of the process scheduler queue, so that it will be run next.

10. Appendix

Summary of Major Modifications to the PDP-1

A. Additional Instructions

Several new instructions were added to the PDP-1 for operating the index register, initiating I/O functions, and for doing arithmetic operations between the active registers. Operation codes were provided for the meta-instructions.

B. I/O Function Bus

An I/O function bus capable of controlling sixty-four independent peripheral devices was added to replace the standard PDP-1 I/O system. The invoke instruction causes a few control words to be sent to the specified device causing it to be activated. The bus control contains a priority circuit to select the highest priority device if more than one completes at the same time.

C. Addressing Modes

When the index register was installed, several instructions were added to control the addressing modes which determine what combination of index and defer addressing is used in effective address calculations.

D. Memory System

The original memory was replaced by 3 four thousand word modules similar to the ones used on the PDP-6 and PDP-10 machines. A core rename (CR) register was added to the CPU so that the modules could be used as pages in the time-sharing system.

Each memory may be accessed independently through any of four "ports." The highest priority port on each module is used for the drum (512K), the next highest for the data channel (which in turn has eight ports), and the two lowest for CPU's (although only one CPU is currently implemented).

E. Mode Structure

When the machine was rebuilt (summer,]966), three distinct hardware modes were implemented: user mode, executive mode, and debug mode. The user and executive mode have been discussed in the body of the paper. Debug mode is superior to executive mode and, while the machine is in this state, all instructions are legal and bad operation codes will stop the machine.

A change to the next higher mode (user to executive or executive to debug) happens when a trap or interrupt (in the first case) occurs. Interrupts remain pending while the computer is in executive mode. A mode change or "break" is accompanied by the storing of all active registers at the location in the executive memory specified by the process pointer (PP) register. When debug mode is entered, the executive state word is deposited in fixed locations. At the end of the break, the index register is left pointing to the newly deposited state word.

In order to descend to the next lower mode, the unbreak instruction was implemented. Unbreak is a many-cycle instruction which does just the opposite of a break -- it completely restores the state of the CPU for a process in the next lower mode.

F. Scheduling Hardware

Two registers, the current priority (CP) and the queue priority (QP), were added. The QP contains the priority of the highest priority process in the queue, while CP contains the priority of the process ehich is actually running. The number in CP controls the number of subquanta allotted to the running process. If QP is less than CP, there is a process of higher priority waiting to be run, and a preempt interrupt will occur at the end of the current subquantum (i.e., within 1 1/4 milliseconds). Otherwise, one of two actions will be taken after the end of the specified number of subquanta: if CP is less than QP, CP will be decremented, the number of allotted subquanta will be reset accordingly, and the process will continue running; if CP is equal to QP, a round robin interrupt will occur.

11. Acknowledgements

The authors wish to thank Professor J. B. Dennis, who did most of the hardware design and specification of the software for the PDP-1 system. His assistance was invaluable in preparingthis paper. Leo Rotenberg, one of Professor Dennis's graduate students, also contributed to the initial design of the system.

John MacKenzie, Ralph Butler, and John Connolly are thanked for their many hours of documentation, wiring, and debugging of the hardware.

12. References

1. Dennis, J. B., Van Horn, Earl C., "Programming Semantics for Multiprogrammed Computations", Comm. of the ACM, vol. 9, no. 3, March, 1966.

2. Dennis, J. B., "A Multiuser Computation Facility for Education and Research", Comm. of the ACM, vol. 7, no. 9, Sept., 1964.

3. Corbato, F. J., et. al., "An Experimental Time-sharing System", AFIPS Conf. Proc. 21, Spartan Books, Baltimore, p. 335 ff.

4. Dijkstra, E. W., Cooperating Sequential Processes, Dept. of Math., Technological University, P.O. Box 513, Eindhoven, The Netherlands.

The following are unpublished internal memoranda:

1. Rotenberg, Leo J., "An Implementation of an Almost Segmented Time-sharing System".

2. Dennis, J. B., Input/Output in the PDP-1-X, no. PDP-33, March 10, 1966.

3. Dennis, J. B., Rotenberg, Leo J., PDP-1-X Index Register, no. PDP-34, April 27, 1966.

4. Modes, Registers, New Instructions and Traps and Interrupts in the PDP-1-X, no. PDP-38, June 7, 1966.